The First Decade of GUI Ripping: Extensions, Applications, and Broader Impacts

(Invited Paper)

Atif Memon, Ishan Banerjee, Bao N. Nguyen, Bryan Robbins
Department of Computer Science,
University of Maryland,
College Park, MD 20742, USA
{atif, ishan, baonn, brobbins}@cs.umd.edu

***Abstract*—This paper provides a retrospective examination of GUI Ripping—reverse engineering a workflow model of the graphical user interface of a software application—born a decade ago out of recognition of the severe need for improving the then largely manual state-of-the-practice of functional GUI testing. In these last 10 years, GUI ripping has turned out to be an enabler for much research, both within our group at Maryland and other groups. Researchers have found new and unique applications of GUI ripping, ranging from measuring human performance to re-engineering legacy user interfaces. GUI ripping has also enabled large-scale experimentation involving millions of test cases, thereby helping to understand the nature of GUI faults and characteristics of test cases to detect them. It has resulted in large multi-institutional Government-sponsored research projects on test automation and benchmarking. GUI ripping tools have been ported to many platforms, including Java AWT and Swing, iOS, Android, UNO, Microsoft Windows, and web. In essence, the technology has transformed the way researchers and practitioners think about the nature of GUI testing, no longer considered a manual activity; rather, thanks largely to GUI Ripping, automation has become the primary focus of current GUI testing techniques.**

I. Introduction

System functional testing—testing the system as a whole—of a software *system under test* (SUT) that contains a graphical-user interface (GUI) front-end requires that test cases, represented as sequences of *events*, be executed on the SUT and its functional correctness verified for each test case [1], [2], [3], [4], [5]. The events are actions that may be performed by a user on the GUI, e.g., clicking on a button, entering text in a text-field, rotating a mobile device, clicking on the *Back* button in a browser, or using the *swipe gesture* on a tablet. Because end-users perform *sequences* of events on a GUI-based application, so too must system testing in order to evaluate the SUT's compliance with its specified requirements.

Before our first report of the design and development of the *GUI Ripping* technology 10 years ago [6], the term *GUI test automation* had a very specific meaning – "automatically executing *manually developed* test cases." That is, a tester would create GUI test cases *by hand*; these test cases would be automatically executed by a test harness on the SUT [7]. Test cases would be programmed or coded using platform-specific libraries such as JFCUnit[1] or WebDriver.[2] Or test cases would be captured/recorded using *capture/replay* (record/playback) tools such as WinRunner[3] (product has been renamed since) and Rational Robot.[4] Because of the resources required for manual creation of GUI test cases, testers would end up with very few test cases, leading to inadequate testing.

Much has changed for GUI test automation in the last decade, largely due to the GUI ripping technology that has been an enabler for research in this field. Numerous advances in model-based GUI testing have been reported [8], [3], [9], [4]. Most importantly, GUI ripping has caused a *semantic shift* for researchers and practitioners. They have now come to view *GUI test automation* as automatic *creation* of test cases, not just their automatic execution.

Our own group at Maryland has leveraged the GUI Ripper and its associated workflow for model-based GUI testing to develop techniques to automatically generate millions of test cases for real-world SUTs. With the Ripper's ability to reverse engineer GUI models as a foundation, we have developed a broader framework for GUI testing as enabled by the Ripper's support for constructing models of GUI applications [10], [11], [12]; conducted large-scale empirical studies to investigate a number of impactful questions relevant for software testing as a whole [13], [14], [15], [16], [17]; developed and evaluated enhanced models and workflows for automated test case generation and execution as applied to the challenges of smoke and regression testing [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [30], [34], [35], [31], [36]; and addressed various additional challenges in model-based testing, such as test suite reduction [37], [38], similarity [39], and meta-analysis [40]. We discuss some of these extensions and applications in Section III.

The GUI Ripper technology has been embraced by other researchers and extended for multiple studies that span a broad range of research domains; including extending the GUI ripping algorithm for multiple platforms [41], [5], [42], [43], [44],

[1]http://www.jfcunit.sourceforge.net
[2]http://www.seleniumhq.org/projects/webdriver
[3]http://www.hp.com/functionaltesting
[4]http://www.ibm.com/software/awdtools/tester/robot

[45], [46], developing new GUI models [47], [48], [49], [50], improving GUI space exploration strategies [51], [52], [53], using reverse engineering for non-GUI testing [54], [55], [56], [57], [58], and developing experimentation benchmarks [52], [59], [60]. We summarize these broader impacts in Section IV.

The GUI Ripper has been realized as a tool in the GUI Testing frAmewoRk (GUITAR) open-source project,[5] hosted on SourceForge.net. In addition to the GUI Ripper, GUITAR contains a suite of tools—most enabled by the GUI Ripper—for automated model-based GUI testing. We elaborate on these tools in Section V.

The GUI ripping technology has led to several Government-sponsored research projects: *Enhancing Testing Techniques for Event-driven Software Applications* (CCF-0447864) that resulted in the development of core algorithms and tools for GUI testing, *COMET-COMmunity Event-based Testing* (CNS-0855055) to investigate the requirements for a community infrastructure of event-based testing researchers to provide uniformity in experimentation, and *COMET: A Web Infrastructure for Research and Experimentation in User Interactive Event Driven Testing* (CNS-1205501) that aims to reduce *experimental mismatch* in testing and advance user interactive event driven testing research by developing a shared research and experimentation web infrastructure called COMET. Our most recent findings from these projects suggest the constant need to improve the GUI ripping technology in the face of new human-computer interaction modalities and diverse sets of platform configurations [61].

II. Overview of the GUI Ripper

At a high level, the GUI ripping technology takes as input an executing GUI-based application and produces, as output, its workflow model(s). We first discuss the models and then describe the algorithms used to obtain them.

A. GUI Models

GUIs have a common property of possessing a hierarchical structure. Microsoft Windows and modern Linux distributions use GUIs consisting of windows that contain widgets. Invoking a widget may create more windows with widgets. Web-based GUIs consist of a web-page with graphical components such as hypertext. Invoking a hypertext in turn leads to more web pages. Modern GUIs on mobile platforms (such as Android-based *apps*) consist of *activities*, each containing widgets which may create more activities.

The invocation of one GUI container, from a widget in another container, is the basis for the hierarchical structure of most GUI-based applications. The original design of the GUI Ripper [6] was based on this hierarchical nature of contemporary GUIs. This design has proven flexible enough to work with today's GUIs that continue to be hierarchical. The GUI Ripper was initially implemented for Java SWT-based GUIs. It was subsequently extended to web-based GUIs, iOS, Android, Java JFC, Java SWT (Eclipse) and UNO (Open Office) frameworks. The core of the GUI Ripper for each platform is based on the original design.

[5]http://guitar.sf.net

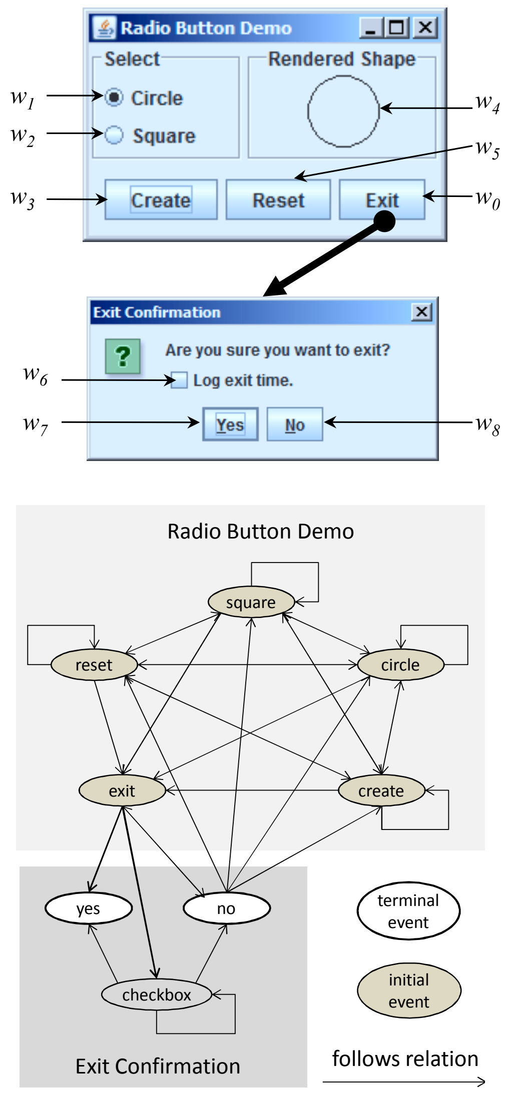

Fig. 1: GUI Tree and Event-Flow Graph for the Radio Button Demo application.

Two models of the GUI are created by the GUI Ripper – **GUI Tree** and **Event-Flow Graph**. The GUI Tree represents the hierarchical nature of the GUI. Each node of the GUI Tree represents a window in the GUI. A window consists of:

- $\mathcal{W} = \{w_1, ..., w_l\}$, a set of widgets contained in the GUI window – for example, $\{$*OK*, *Cancel*, *Save*$\}$.
- $P = \{p_1, .., p_m\}$, a set of properties for each widget in $\mathcal{W}$ – for example, $\{$*width*, *color*$\}$.
- $V = \{(v_{11}, ..., v_{1m}), ..., (v_{l1}, ..., v_{lm})\}$, a set of values for each property of each widget – for example, $\{$(100, black), (200, grey), (300, white)$\}$.

One GUI window at any instant can be described by its state, given by the triple $\{\mathcal{W}, P, V\}$. The GUI Tree is composed of all windows in the GUI. It is represented by the triple

$\{\Psi, \tau, \epsilon\}$, where Ψ is the set of all windows $\{W_1, ..., W_n\}$ in the GUI, $\tau \subseteq \Psi$ is a set of top-level windows and $\epsilon = \{(x_1, y_1), ..., (x_e, y_e)\}$ is a set of e directed edges. A directed edge exists from a window x to a window y if an event executed on x causes the SUT to create the window y.

GUI windows can be categorized into two types, based on widgets that are executable when that window is created – *modal* and *modeless* windows. When visible, modal windows force the user to execute an event from within the modal window – for example, the `Save` window in Microsoft's WordPad software. Modeless windows, on the other hand, expand the set of executable events on the existing GUI – for example, the `Replace` window in Microsoft's WordPad.

The Event-Flow Graph (EFG) represents the sequences of executable events on the GUI of an SUT. A node in the Event-Flow Graph is an *event* executable on the GUI. An edge $e_x \rightarrow e_y$ from event x to y indicates that event y is executable *immediately* after executing event x. The edge models the *follows* relation between events x and y. The Event-Flow Graph of the GUI of an SUT may be inferred from its GUI Tree. This is done by identifying an executable widget, represented as event x, on a window W and identifying the set of all executable widgets on all visible windows, after executing x. The Event-Flow Graph will contain an edge from x to all events in this set.

Figure 1 shows the example of a toy Java-JFC based application called Radio Button Demo. The figure on the top shows the GUI Tree of the application while the bottom figure shows its Event-Flow Graph. The GUI Tree consists of two windows, `Radio Button Demo` whose `Exit` button invokes the `Exit Confirmation` modal window. The EFG shows the initial, terminal events and the *follows* relation between events.

B. Ripper Algorithm

The GUI Ripper automatically creates a model of the GUI of an application-under-test (SUT) from its executable binary form. The GUI Ripper launches the SUT, identifies its initial GUI windows, automatically extracts each window's GUI properties, widgets and their properties, automatically invokes each widget and repeats the process recursively for new windows that may be created. The GUI Ripper essentially traverses the hierarchical GUI in a top-down manner, extracting properties. These properties and the observed hierarchical relationships between windows are used to construct models of the GUI.

The steps executed by the GUI Ripper are given by the procedure `RIP` in Figure 2. In this procedure, the SUT is launched in line 1, and its top-level GUI windows are identified; in lines 3–4, the procedure `RIP-RECURSIVE` operates on each top-level GUI window. The `RIP-RECURSIVE` procedure acts on a given GUI window. In line 5, it extracts all widgets and their properties using platform-dependent APIs; in line 6, it identifies a subset of extracted widgets which are executable; for each executable widget, line 8 executes the widget's event; line 9 detects if additional GUI windows were created due to the widget execution; and line 10 updates the set of GUI windows of the SUT. Lines 10–11 recursively process the invoked windows C.

```
$GUI = \phi$
PROCEDURE RIP (SUT $A$)
  $\tau$ = get-top-level-windows(A)                    1
  $G = \tau$                                           2
  FORALL $t \in \tau$ DO                               3
    RIP-RECURSIVE($t$)                                 4
PROCEDURE RIP-RECURSIVE (Window $t$)
  $\Psi$ = extract-widgets-and-properties($t$)         5
  $\epsilon$ = identity-executable-widgets($\Psi$)     6
  FORALL $e \in \epsilon$ DO                           7
    execute($e$)                                       8
    $C$ = get-invoked-windows($e$)                     9
    $GUI = GUI \cup C$                                 10
    FORALL $c \in C$ DO                                11
      RIP-RECURSIVE($c$)                               12
```

Fig. 2: Pseudo-code for the GUI Ripper. A hierarchical GUI is recursively extracted from the executable binary of the SUT A. GUI contains the resulting GUI structure.

The procedure `RIP` creates the GUI Tree that represents structural information about the GUI, such as (1) the set of GUI windows, (2) the set of widgets and their properties in each GUI window, and (3) the set of GUI windows appearing after executing a widget.

Semantic information between widgets is inferred from the GUI Tree and stored in the Event-Flow Graph. For a given widget *w* in the GUI Tree, the set of widgets, $\{w_1 ... w_n\}$, that are executable *immediately* after executing *w* are identified and edges $\{w \rightarrow w_1, ..., w \rightarrow w_n\}$ are added to the EFG.

III. Ripper as a GUI Testing Enabling Technology

The introduction of GUI ripping methodology and its associated tools has enabled a large body of research on GUI testing as a whole over the past decade. First, we focus on the line of research at the University of Maryland which directly extends from the original GUI ripping work.

Figure 3 highlights the key role played by GUI ripping in a number of model-based GUI testing techniques developed at the University of Maryland since its inception. The standard workflow from the original work produces artifacts of a GUI tree, EFG, test cases, and test results, in that order. This workflow has been expanded in two major ways. Some techniques now incorporate a feedback loop to iteratively enhance models based on data collected during test case replay. Moreover, some models collect user profiles as an alternative method of reverse engineering. Below, we expand on the development of these techniques and their research results.

A. Open-Source Tools for Testing

Perhaps the most visible results from the GUI ripping work come in the form of software artifacts. The testing workflow of GUI ripping, model construction, test case generation, and

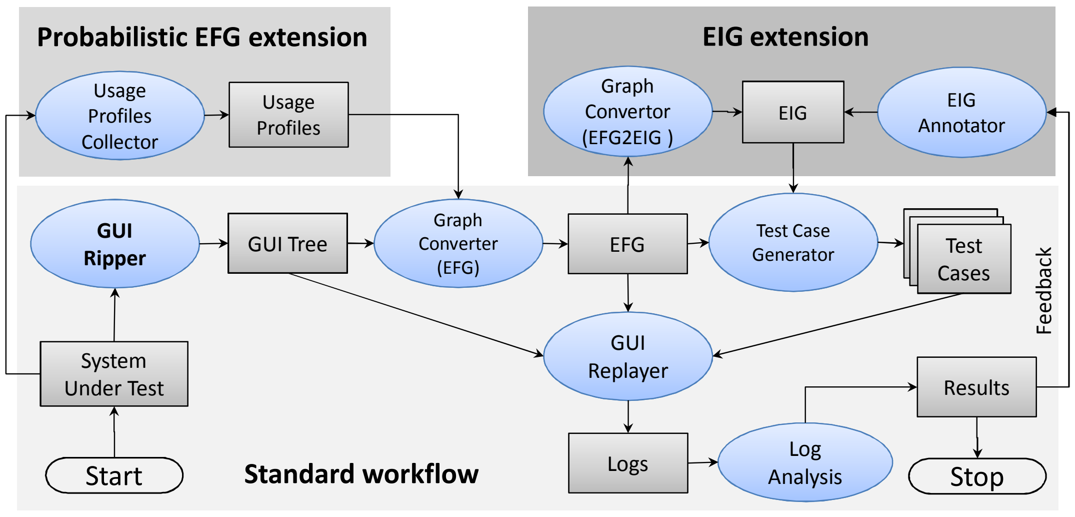

Fig. 3: Model-based GUI testing techniques enabled by the GUI Ripper.

test case replay provide the basis for GUITAR, an open-source framework for model-based GUI testing developed at the University of Maryland and now available for a growing number of application platforms [10]. The GUITAR tools directly implement the GUI ripping methodology. Researchers and practicing testers can leverage GUITAR to perform model-based testing as enabled by the Ripper.

Architecturally, GUITAR supports plugins at every phase of its default model-based GUI testing process: ripping, model construction, test case generation, and replaying. GUITAR supports customization of widgets (e.g., tabs in a window) events (e.g., text entry), data collection (e.g., code coverage and error logs) and entire application platforms (e.g., Web, Mobile [11], SWT, UNO, and others) by supporting Java extensions to its Ripper and Replayer tools. Similarly, customizations to the model construction and test case generation process [12] support new models and coverage criteria, respectively.

Over the years, we have performed a large body of research by extending GUITAR and applying customized tools to real-world applications. A great deal of this work has only been possible due to the Ripper's ability to reverse engineer useful models of applications in an automated way.

B. Support for Large-scale Empirical Studies

GUITAR's plugin-based architecture and ability to integrate with other test harnesses has helped researchers conduct large-scale experiments investigating important GUI testing activities.

Xie et al. examine the characteristics of a "good" GUI test suites, considering the effects of event-composition, test case length, and event context on test case execution and effectiveness [13]. This foundational work would inform a great deal of future work on incorporating event context into GUI testing techniques (see, for example, the model and workflow enhancements with this same goal outlined below). Later, Xie et al. [14] used GUITAR to investigate the effects of GUI oracle specificity, proposing 6 types of GUI oracles and using a series of experiments on four fault-seeded Java applications to evaluate their strengths and weaknesses. Their work showed that choice of oracle type indeed has large effects on fault detection ability, which not only highlights the importance of addressing oracle specification in automated model-based testing, but also reinforces the differences between a test suite's ability to meet coverage criteria and its ability to find faults.

McMaster and Memon developed and evaluated an improved notion of fault detection capability to support later experimentation with test suite reduction techniques [15]. Similarly, Strecker et al. [16], [17] leveraged GUITAR to investigate the relationships between testing techniques and the characteristics of faults detected. This very large scale empirical study involved the execution of 100 test suites on 2 fault-seeded open-source applications, consuming nearly 100 machine-days and offering a number of important conclusions about proper evaluation of empirical results in software testing experiments.

C. GUI Testing Workflows

A number of empirical investigations carried out within our group focus on the evaluation of specific techniques for GUI testing. These techniques leverage and improve upon the GUI ripping methodology's original automated testing workflow.

In 2005, Memon et al. proposed the DART QA process for rapidly evolving software [18], [19]. DART uses a Ripping-first workflow very similar to the original Ripping methodology we proposed, as supported by GUITAR tools. DART aims to automate regression testing tasks, including model construction, test case generation, and analysis of test results. Xie and Memon proposed the use of a Ripper-first workflow for crash testing of rapidly evolving activities [20], noting that automated crash testing is capable of detecting interesting bugs. Memon and Xie further concluded that the DART workflow can be applied to daily smoke testing tasks, concluding that more advanced oracles can be used to compensate for any lack of longer test cases or large numbers of test cases [21]. Later, the authors showed that a multi-paradigm approach of crash testing, smoke testing, and comprehensive GUI testing can support open-source software projects with distributed teams [22].

Yuan and Memon developed the ALT workflow, which progressively improves test cases by alternating between test case generation and execution iteratively [23], [24]. Memon also developed a workflow which included automated detection and repair of unusable test cases, a common problem in the model-based regression testing of GUI-based software as supported by the original Ripper methodology [25]. Later, Huang et al. developed a similar tool employing genetic algorithms for repair instead [26]. Yuan et al. also similarly proposed an evolutionary algorithm for test case repair [27], and Cohen et al. proposed the combination of test case repair with covering arrays to address incomplete specification in GUI models [28].

McMaster and Memon approached the problem of GUI test case maintenance with a heuristic-based framework, which used an approach similar to the GUI Ripper to build and compare models of applications [29].

Enhanced workflows often work in combination with customized GUI models to improve GUI testing approaches. The EIG and ESIG models (described in more detail below) both incorporate some type of test case feedback (i.e., data collected from the execution of test cases) to improve on the basic EFG [30], [31]. The grey-box testing technique of Arlt et al. augments Ripper output with information from a data dependency model to improve test case generation [32].

D. Enhanced Models

The event-flow graph (EFG) proposed in the original GUI Ripper work has proven to be a very fundamental model in much of our research on model-based testing since that time. In 2007, Memon more formally outlined the event-flow model [33]. Test case generation operating over the event-flow model is now a core component of the GUITAR framework and its standard Ripper-first workflow as well, with a series of Model Converters applied to convert Ripper output into a desired model prior to test case generation. Researchers have developed and evaluated a number of alternative models, based largely on the EFG, which show improved testing capability.

One line of research in model-based GUI testing involves improving upon the EFG directly to identify events and event relationships most likely to find faults. Xie and Memon developed the event-interaction graph (EIG), which focuses on identifying and leveraging within test cases *interacts-with* relationships between events [30]. The *interacts-with* relation leverages the Ripper output's knowledge of modal window terminating events, which the authors show greatly impacts the fault detection capability of event sequences. The authors also demonstrated automated model construction and test case generation algorithms for the EIG which build on the Ripper's reverse engineering capabilities. Yuan et al. later develop a covering array model for GUI test case generation [34] and several additional coverage criteria which improve upon the standard test case generation methods originally applied to the EIG [35].

Similar to the EIG, the event-semantic interaction Graph (ESIG) model described by Yuan and Memon [31] further reduces the EIG by supplementing its structural information from the Ripper with semantic relationships learned through test case execution. The ESIG captures and leverages an event-semantic interaction (ESI) relation also shown to be more effective at detecting faults in GUI-based applications.

Brooks and Memon incorporated actual usage data for an application into a *probabilistic EFG*, which showed the ability to detect faults which were undetected by user sequences alone, even with small test suites [36].

E. Additional Support for Model-based Testing

In addition to the enhanced workflows, models, and their associated coverage criteria outlined above, we have also used the Ripper to investigate additional techniques and factors in model-based testing.

McMaster and Memon developed and evaluated a test suite reduction technique based on an adequacy criteria of call-stack coverage [37], [38]. This technique involved collecting call stack information during test case execution and showing that test cases which duplicate existing call stack coverage do not tend to contribute to fault detection capability.

Bryce and Memon showed the application of event-interaction coverage to test suite prioritization, evaluating existing test suites on the basis of their coverage of *interacts-with* relations from a model. Brooks and Memon developed a test suite similarity metric, *CONTeSSi*, based on the consideration of event context when comparing test cases [39]. Elsaka et al. applied network analysis techniques to models, discovering various techniques for test case prioritization and for the decoupling of tests [40].

IV. Ripper's Broader Impacts

Research enabled by the GUI Ripper has also effectively impacted efforts beyond our research at the University of Maryland, as other researchers continue to assimilate GUI Ripping into their GUI testing work. Since our original work, others have done related work which (1) extends the GUI Ripping algorithm to additional platforms, (2) develops alternative, EFG-like GUI models, (3) improves the GUI space exploration strategy used by the GUI Ripper, (4) applies similar reverse engineering outside of the GUI-testing domain, and (5) develops benchmarks for empirical research.

A. Extending GUI Ripping to Other Platforms

Our original GUI Ripping algorithm [6] was implemented only for Java Swing applications. Based on this work, other researchers have implemented this algorithm for additional GUI platforms outside of our work in developing GUITAR.

Mesbah et al. [41], [5] extend the concept of GUI Ripping to crawl Rich Internet Applications (RIAs). Unlike a traditional web page, which is typically static, a single RIA page may consist of multiple *states*. On the client side, a user can transform between these states by performing events on the GUI. Hence, a simple hypertext-based algorithm for discovering the GUI hierarchy cannot be directly applied. To solve this problem, the authors develop a tool called Crawljax. Crawljax identifies all clickable elements on the website and triggers clicks on them to automatically reveal different hidden web elements. Similar in nature to the GUI Ripper, Crawljax traverses a website in a depth-first order until no new GUI elements are found. The result is analyzed to infer a state machine model called a state-flow graph. Duda et al. [42] propose a similar approach, but use a breadth-first strategy to support parallel crawling and speed up the reverse engineering process.

Amalfitano et al. [43], [62] use the concept of ripping to reconstruct a finite-state machine model representing its behaviors from multiple execution traces. The reconstruction process consists of two steps. First, the RIA is executed, by the user, in a controlled environment to capture execution event sequences in different usage scenarios. Then, from these event sequences, an abstract FSM is derived by leveraging a data clustering technique.

The GUI ripping idea is also extended for mobile applications. The main challenge to performing traditional model-based testing for mobile applications is the requirement of working in heterogeneous environments, where UI automation across in all possible environments is often not trivial or feasible to maintain. Joorabchi et al. [44] implement a tool called iCrawler to automatically reverse engineer the structure of the GUI of an iPhone operating system (iOS) application. The authors use a low-level Objective-C runtime reference library to hook into the executing iOS application. iCrawler automatically sends events to the user interface to cover the interaction state space, resulting in a state model representing all possible GUI states of the application. Amalfitano et al. [63], [45] apply a similar approach for Android applications. However, instead of directly accessing low-level functions to interact with the GUI, they use a higher-level automation library called Robotium.[6]

B. Developing EFG-like Models

Researchers have enhanced the GUI Ripper's models with additional information to implement specific use cases. Huang et al. [47] introduce a *weighted EFG* model for test case generation. Testers first use the GUI Ripper to obtain a non-weighted version of the EFG. Then, based on domain knowledge of the SUT, the testers manually assign weights for each node in the EFG. With the enhanced EFG, test case generation focuses on weighted portions of the GUI. An empirical study with 3 open-source applications showed that this method can obtain a better fault detection rate than the standard, EFG-only workflow.

Huang et al. [48] develop an automatic strategy to dynamically assign weights for the GUI Ripper's models. An "ant colony" algorithm is applied to dynamically adjust the weights on-the-fly during test case execution.

To complement the dynamic approach of the GUI Ripper, Arlt et al. [46] perform a light-weight static analysis of the source code of the GUI application to extract code-level dependencies between GUI event handlers. The inferred dependencies are used to construct a new model called Event Dependency Graph (EDG). Using this model, the authors are able to generate better test cases to find previously undetected bugs in four open-source Java applications.

The GUI Ripper's model has also been tailored to work with mobile applications. Yang et al. [49] proposed an improved EFG model for the Android event system. They apply static analysis to the Java source code of Android apps to detect actions associated with each GUI state. They enrich the EFG with this information to enable better test case generation. This model is implemented in a tool called ORBIT and evaluated against 8 Android applications. Tanzirul et al. [50] propose a similar approach, but the information used to enrich the EFG is extracted from the bytecode instead of source code.

C. Improving GUI Input Space Exploration

Fundamentally, the GUI Ripper uses dynamic analysis to model a GUI's input space. This idea has been explored and expanded by several researchers. Paiva et al. [64] [65] propose a series of semi-automated techniques to incrementally reverse engineer the GUI structure. A skeleton of a state machine model is first obtained by automatically exploring the GUI to discover as much as possible the GUI structure and behaviors. This automatic exploration process is augmented with manual exploration to enable accessing parts of the GUI that are not accessible automatically. The combined model is validated manually by the tester. This approach of interleaving automated and manual reverse engineering helps minimize model creation effort while incorporating the domain knowledge, of the human tester into the final model. AutoBlackTest [52] and EXSYST [66] are two recent tools which apply machine learning techniques to learn test case generation models. The testing process starts by executing a small, manually-created seed test suite. The results from test executions are then used to infer a state machine to generate additional test cases.

Other authors also leverage a variety of reverse engineering techniques to construct the GUI model and enable automated GUI testing. These techniques include static analysis [67], symbolic execution [68], concolic testing [69], model-checking [70], and computer vision [71].

D. Reverse Engineering for Non-GUI Testing

Motivated by the idea of reverse engineering for testing, researchers in other domains have also proposed testing techniques based on reverse engineering. In unit testing, Pacheco et al. [54] develop a random testing technique for object-oriented applications. Their technique starts by dynamically reverse engineering the list of all possible method calls of the class under test. Test cases are then generated by randomly combining the method calls into sequences. As the test cases are executed, feedback is collected to incrementally infer relations between methods. These inferred relations are used to avoid generating test cases which are not executable. Zhang et al. [55] improve this technique further by incorporating static analysis

[6]http://code.google.com/p/robotium/

techniques to effectively select parameter values. Dallmeier et al. [56] propose a similar technique but in a more systematic way. By analyzing the execution traces from previous test cases, they incrementally learn a state machine model representing the class under test. This state machine is then used to generate additional, complementary test cases.

Wang et al. [57] apply a similar idea to automatically generate test cases for context-driven applications, such as those in cell phones, PDAs, and portable consoles. Due to the high complexity level of these applications, adequately identifying and developing models of their behaviors is difficult. The authors propose a new adaptive behavior model called Adaptation Finite State Machine (A-FSM). This model is incrementally refined during the application's execution. With an A-FSM, the authors were able to automatically generate test cases to cover a large spectrum of the application's behaviors. An empirical study of the A-FSM model identified a number of fault patterns describing classes of faulty behaviors in context-driven applications [72].

Swearngin et al. [58], [73] use the GUI Ripper to construct a model for *predicting human performance* in Human-Computer Interaction studies. They implement a tool called CogTool-Helper that leverages the GUI Ripper as a back-end component.

Researchers have also developed similar ideas in other areas, including web compatibility testing [74], automated debugging [75] and validation of legacy software [76].

E. Developing Benchmarks for Empirical Research

Various researchers have also used the GUI Ripper (often combined with other tools from GUITAR) to develop benchmarks for empirical studies. Mariani et al. [52] introduced a new model-based approach to address the new challenges in testing modern GUI applications. The authors replicate a Ripper-first testing workflow on four open-source applications and use the results as a baseline to compare with their own techniques. In a similar effort, Belli et al. [59] use GUI Ripper to evaluate a new event model called the Event-Sequence Graph. A case study of applying the GUI Ripper to two large modules of a commercial web portal ISELTA was conducted to compare their new model to the EFG.

In a similar effort, Michail and Xie [60] use the GUI Ripper to evaluate their research tool called Stablizer, which helps the user effectively when working with unstable GUI applications. The tool monitors a user's actions in the background and gives a warning as well as the opportunity to abort the action, when a user attempts certain unstable actions previously encountered by other users. The authors develop an experimentation benchmark consisting of four fault-seeded open-source applications. To minimize the threats of validity caused by the artificial fault seeding process, the GUI Ripper was used to eliminate what the authors considered to be obvious and easy-to-detect faults in the benchmark.

V. Using the Ripper for Testing: A Use Case

We now demonstrate a simple use case for the GUI Ripper as implemented within the GUITAR model-based testing framework. Following from the original GUI ripping methodology, GUITAR (1) uses *reverse engineering* to automatically create the structural and semantic model of a GUI from the executable binary of the SUT, (2) automatically generates test cases using different test case generation methods, and (3) automatically executes these test cases on the GUI. As enabled by the GUI Ripper, GUITAR does not require a specification of the GUI from the user, but works with the executable binary of a SUT.

GUITAR has been extended for multiple GUI application platforms, including Java (JFC, SWT), Web, iOS, Android and UNO. GUITAR can be integrated into automated testing harnesses to satisfy GUI testing requirements of real-world applications. Below, we describe a simple GUI regression testing workflow which uses GUITAR, known as GUITAR's *standard workflow*.

The standard workflow, shown in Figure 3, consists of four GUITAR components – the *GUI Ripper*, *Graph Converter*, *Test Case Generator* and *Replayer*. This workflow of GUITAR can work with GUI applications on different platforms. To work with a specific platform, platform-specific *plugins* are required for the GUI Ripper and Replayer. Similarly, the Graph Converter and Test Case Generator can be used to derive different semantic models and use different test-case generation algorithms. A *toolchain* can be created by combining these four components as customized for platform-specific and algorithm-specific scenarios.

A JFC toolchain for testing JFC-based applications is developed by first combining JFC-specific plugins for the GUI Ripper and Replayer to create the *JFCGUIRipper* and the *JFCGUIReplayer*. These tools use the Java Accessibility framework to extract widgets and their properties from the GUI of a Java Swing application. A standard Graph Converter, the EFGConverter, produces an EFG from a GUI tree, creating the EFGConverter. Similarly, a standard Test Case Generator, SequenceLength Generator, provides test case generation from an EFG. The SequenceLength Generator generates all possible test cases of a given length (L), prefixing a path to an initial event if necessary. In terms of test oracle, a simple script-based analysis of the JFCReplayer's logs provides a simple oracle to identify SUT crashes during replay, which are assumed to have revealed faults.

We recently applied the standard workflow to two popular JFC-based applications – ArgoUML[7] and JabRef.[8] Supported by the JFC toolchain described above, the JFCGUIRipper's structural information was converted into an EFG, allowing the generation and replay of all event sequences of length 2 in the resulting model. After initial configuration of the SUT, this process proceeded in a fully automated manner. A machine with single core, 2GHz CPU and 1GB memory running Redhat Enterprise Linux 5 was used for ripping, graph conversion and test case generation. A cluster of the same identical machines was used for replaying.

Table I(a) shows information retrieved at each of the four steps of the standard workflow. From this table, both ArgoUML

[7]http://argouml.tigris.org

[8]http://jabref.sourceforge.net

and JabRef are non-trivial applications, with 69K and 44k lines of code. The GUI Ripper extracted 30 and 40 windows, 1548 and 1285 widgets respectively. The JFCGUIRipper was configured to ignore some of each SUT's windows to avoid complications such as unknown widgets or events and unwanted side effects (e.g., printing). The EFGConverter produced EFGs with 328 and 376 nodes, and 4468 and $15,652$ edges respectively. The SequenceLength Generator then produced 4468 and $15,652$ test cases respectively, matching the number of edges in the EFG. (Note that length-two sequences represent EFG edges).

The JFCReplayer, detected and recorded in a log any exceptions which occurred during test case replay. These logs were automatically analyzed to identify crashes. In all, execution of the test cases detected 3 unique faults in ArgoUML and 4 unique faults in JabRef which could be manually verified as application bugs. Table I(b) lists the faults detected in ArgoUML and JabRef.

VI. Summary and Conclusions

The GUI ripping technology developed a decade ago was the first to use reverse engineering to build a model of a software system under test (SUT) and subsequently use the model for testing. The idea of "using a software to test itself" has many skeptics, for good reason. After all, how can one test software without using its specifications or use cases? System testing needs to evaluate the SUT's compliance with its specified requirements. The approach of *rip-model-test* enabled by the GUI Ripper is not meant to replace requirements- and specifications-based testing. Rather, it provides an additional tool for the tester's toolbox. We have shown this to be effective at finding faults via multiple empirical studies.

The rip-model-test approach is fully automated, and hence can be used without any human intervention throughout the software development lifecycle to detect catastrophic failures, such as crashes. Once the software's GUI has stabilized, additional test cases may be developed manually to detect bugs due to deviation from requirements.

Our earlier systematic mapping study on GUI testing [77] showed that while there is sufficient interest in model-based GUI testing amongst academics, most popular commercially available tools are not model-based. We feel that this *technology gap* is due to the complexity of model-based techniques; and lack of testers' knowledge of models. The automatic model extraction and use process enabled by the GUI Ripper will ultimately help to bridge the gap between research and practice.

In the last decade, we have developed numerous test automation processes around GUI ripping. We have also implemented numerous tools that we have used to generate millions of test cases and to understand the complex relationship between GUI faults and test cases. The GUI ripping methodology has also been embraced and extended by numerous researchers in ways we never anticipated. We continue to improve the GUI testing technology in various ways: new algorithms for model building, new models, new platforms, and new testing techniques based on the models. We feel that these exciting new developments will yield another decade of fruitful research.

TABLE I: Executing the Standard workflow on ArgoUML and JabRef

		ArgoUML	**JabRef**
Ripping	Lines of code	$69,954$	$44,522$
	Windows	30	40
	Widgets	$1,548$	$1,285$
	Ignored	2	6
	Terminal	13	14
	Time (s)	231	431
Model conversion	Nodes	328	376
	Edges	$4,468$	$15,562$
	Time (s)	4	5
Test case generation	Test cases	$4,468$	$15,562$
	Time (s)	213	875
Replaying	Statement coverage	22.45%	29.12%
	Branch coverage	10.31%	12.04%
	Log size (GB)	8.4	185.0
	Fault detected	3	4
	Time (hours)	309	$1,204$

(a) Intermediate result of executing the standard workflow on ArgoUML and JabRef.

Faults detected	
	ArgoUML
FileNotFoundException	*File* → *Export Graphic* → filename: `invalid filename` → *Save*
FileNotFoundException	*File* → *Export All Graphics* → filename: `invalid filename` → *Save*
Incorrect log message	*Edit* → *Delete from Model*
	JabRef
FileNotFoundException	*Option* → *Manage journal abbreviation* → new filename: `invalid filename` → *Ok*
MalformedURLException	*Option* → *Journal abbreviation* → *Download* → URL: `invalid URL` → *Ok*
NullPointerException	*Option* → *Manage custom imports* → *Add from folder* → folder path: `non-existing folder` → *Cancel*
ZipException	*Option* → *Manage custom imports* → *Add from jar* → filename: `non-existing filename` → *Select a Zip-archive*

(b) List of faults detected in ArgoUML and JabRef using the standard workflow.

Acknowledgments

This work was partially supported by grant number CNS-1205501 by the US National Science Foundation (NSF), and Project Number 1R01AI100947-01A1 by the US National Institute of Allergy and Infectious Diseases (NIAID), part of the US National Institutes for Health (NIH).

References

[1] A. M. Memon, "A comprehensive framework for testing graphical user interfaces," Ph.D., 2001, advisors: Mary Lou Soffa and Martha Pollack; Committee members: Prof. Rajiv Gupta (University of Arizona), Prof. Adele E. Howe (Colorado State University), Prof. Lori Pollock (University of Delaware).

[2] Lee White and Husain Almezen, "Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences," in *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2000, p. 110.

[3] J. L. Silva, J. C. Campos, and A. C. R. Paiva, "Model-based user interface testing with Spec Explorer and ConcurTaskTrees," *Electron. Notes Theor. Comput. Sci.*, vol. 208, pp. 77–93, 2008.

[4] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Softw. Test. Verif. Reliab.*, vol. 17, pp. 137–157, September 2007.

[5] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-Based Automatic Testing of Modern Web Applications," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 35–53, 2012.

[6] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, ser. WCRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 260–.

[7] A. M. Memon, "GUI testing: Pitfalls and process," *Computer*, vol. 35, no. 8, pp. 87–88, 2002.

[8] S. Dalal, A. Jain, C. Lott, G. Patton, N. Karunanith, J. M. Leaton, and B. M. Horowitz, "Model-Based Testing in Practice," in *Proceedings of the 21st International Conference on Software Engineering*. ACM Press, May 1999, pp. 285–294.

[9] T. Pajunen, T. Takala, and M. Katara, "Model-Based Testing with a General Purpose Keyword-Driven Test Automation Framework," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, march 2011, pp. 242 –251.

[10] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: an innovative tool for automated testing of GUI-driven software," *Automated Software Engineering*, pp. 1–41, 2013.

[11] D. Amalfitano, A. R. Fasolino, S. D. Carmine, A. Memon, and P. Tramontana, "Using GUI Ripping for Automated Testing of Android Applications," in *ASE '12: Proceedings of the 27th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2012.

[12] D. Hackner and A. M. Memon, "Test Case Generator for GUITAR," in *ICSE '08: Research Demonstration Track: International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2008.

[13] Q. Xie and A. M. Memon, "Studying the Characteristics of a 'Good' GUI Test Suite," in *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006)*. IEEE Computer Society Press, Nov. 2006.

[14] Q. Xie and A. M. M. Memon, "Designing and comparing automated test oracles for GUI-based software applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, p. 4, 2007.

[15] S. McMaster and A. M. Memon, "Fault Detection Probability Analysis for Coverage-Based Test Suite Reduction," in *ICSM '07: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'07)*. Paris, France: IEEE Computer Society, 2007.

[16] J. Strecker and A. M. Memon, "Relationships Between Test Suites, Faults, and Fault Detection in GUI Testing," in *ICST '08: Proceedings of the First international conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2008.

[17] J. Strecker and A. M. Memon, "Accounting for Defect Characteristics in Evaluations of Testing Techniques," *ACM Trans. on Softw. Eng. and Method.*, 2012.

[18] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 884–896, Oct. 2005.

[19] A. Memon, A. Nagarajan, and Q. Xie, "Automating regression testing for evolving GUI software," *Journal of Software Maintenance and Evolution*, vol. 17, no. 1, pp. 27–64, Jan. 2005.

[20] Q. Xie and A. M. Memon, "Rapid "Crash Testing" for Continuously Evolving GUI-Based Software Applications," in *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 473–482.

[21] A. M. Memon and Q. Xie, "Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 884–896, 2005.

[22] Q. Xie and A. M. Memon, "Model-Based Testing of Community-Driven Open-Source GUI Applications," in *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 145–154.

[23] X. Yuan and A. M. Memon, "Alternating GUI Test Generation and Execution," in *TAIC PART '08: Proceedings of the IEEE Testing: Academic and Industrial Conference 2008*. Washington, DC, USA: IEEE Computer Society, 2008.

[24] ——, "Iterative execution-feedback model-directed GUI testing," *Information and Software Technology*, vol. 52, no. 5, pp. 559 – 575, 2010.

[25] A. M. Memon, "Automatically Repairing Event Sequence-Based GUI Test Suites for Regression Testing," *ACM Trans. on Softw. Eng. and Method.*, 2008.

[26] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing GUI test suites using a genetic algorithm," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 245–254.

[27] X. Yuan, M. Cohen, and A. M. Memon, "Towards Dynamic Adaptive Automated Test Generation for Graphical User Interfaces," in *TESTBEDS'09: Proceedings of the First International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software*. Washington, DC, USA: IEEE Computer Society, 2009.

[28] M. Cohen, S. Huang, and A. Memon, "AutoInSpec: Using Missing Test Coverage to Improve Specifications in GUIs," in *ISSRE'12 Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2012.

[29] S. McMaster and A. M. Memon, "An Extensible Heuristic-Based Framework for GUI Test Case Maintenance," in *TESTBEDS'09: Proceedings of the First International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software*. Washington, DC, USA: IEEE Computer Society, 2009.

[30] Q. Xie and A. M. Memon, "Using a Pilot Study to Derive a GUI Model for Automated Testing," *ACM Trans. on Softw. Eng. and Method.*, 2008.

[31] X. Yuan and A. M. Memon, "Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 81–95, 2010.

[32] S. Arlt, I. Banerjee, C. Bertolini, A. M. Memon, and M. Schaf, "Grey-box GUI Testing: Efficient Generation of Event Sequences," *CoRR*, vol. abs/1205.4928, 2012.

[33] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[34] X. Yuan, M. Cohen, and A. M. Memon, "Covering Array Sampling of Input Event Sequences for Automated GUI Testing," in *ASE '07: Proceedings of the 22nd IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2007.

[35] X. Yuan, M. B. Cohen, and A. M. Memon, "GUI Interaction Testing: Incorporating Event Context," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559–574, 2011.

[36] P. Brooks and A. M. Memon, "Automated GUI Testing Guided by Usage Profiles," in *ASE '07: Proceedings of the 22nd IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2007.

[37] S. McMaster and A. M. Memon, "Call Stack Coverage for GUI Test-Suite Reduction," in *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006)*. IEEE Computer Society Press, Nov. 2006.

[38] ——, "Call-Stack Coverage for GUI Test-Suite Reduction," *IEEE Trans. Softw. Eng.*, 2008.

[39] P. Brooks and A. M. Memon, "Introducing a Test Suite Similarity Metric for Event Sequence-Based Test Cases," in *ICSM '09: Proceedings of the 23rd IEEE International Conference on Software Maintenance*. Alberta, Canada: IEEE Computer Society, 2009.

[40] E. Elsaka, W. E. Moustafa, B. Nguyen, and A. M. Memon, "Using Methods & Measures from Network Analysis for GUI Testing," in *TESTBEDS 2010: Proceedings of the International Workshop on TESTing*

Techniques & Experimentation Benchmarks for Event-Driven Software. Washington, DC, USA: IEEE Computer Society, 2010.

[41] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes," *ACM Transactions on the Web (TWEB)*, vol. 6, no. 1, pp. 3:1–3:30, 2012.

[42] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou, "AJAX Crawl: Making AJAX Applications Searchable," in *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ser. ICDE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–89.

[43] D. Amalfitano, A. Fasolino, and P. Tramontana, "Reverse Engineering Finite State Machines from Rich Internet Applications," in *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, 2008, pp. 69–73.

[44] M. Joorabchi and A. Mesbah, "Reverse Engineering iOS Mobile Applications," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, 2012, pp. 177–186.

[45] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261.

[46] S. Arlt, A. Podelski, C. Bertolini, M. Schaf, I. Banerjee, and A. Memon, "Lightweight Static Analysis for GUI Testing," in *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, 2012, pp. 301–310.

[47] C.-Y. Huang, J.-R. Chang, and Y.-H. Chang, "Design and analysis of GUI test-case prioritization using weight-based methods," *J. Syst. Softw.*, vol. 83, no. 4, pp. 646–659, Apr. 2010.

[48] Y. Huang and L. Lu, "Apply ant colony to event-flow model for graphical user interface test case generation," *IET Software*, vol. 6, no. 1, pp. 50–60, 2012.

[49] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering*, ser. FASE'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 250–265.

[50] T. Azim and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps," in *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2013, to appear.

[51] A. C. R. Paiva, J. C. P. Faria, and P. M. C. Mendes, "Reverse engineered formal models for GUI testing," *Formal methods for industrial critical systems*, vol. 4916, no. 1, pp. 218–233, 2008.

[52] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "AutoBlackTest: a tool for automatic black-box testing," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 1013–1015.

[53] Florian Gross, Gordon Fraser, and Andreas Zeller, "EXSYST: Search-based GUI Testing (Demo Paper)," February 2012.

[54] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84.

[55] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 353–363.

[56] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller, "Automatically Generating Test Cases for Specification Mining," *Software Engineering, IEEE Transactions on*, vol. 38, no. 2, pp. 243–257, 2012.

[57] Z. Wang, S. Elbaum, and D. Rosenblum, "Automated Generation of Context-Aware Tests," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007, pp. 406–415.

[58] A. Swearngin, M. B. Cohen, B. E. John, and R. K. E. Bellamy, "Easing the Generation of Predictive Human Performance Models from Legacy Systems," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '12. ACM, 2012, pp. 2489–2498.

[59] F. Belli, M. Beyazit, and N. Güler, "Event-Based GUI Testing and Reliability Assessment Techniques – An Experimental Insight and Preliminary Results," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 212–221.

[60] A. Michail and T. Xie, "Helping users avoid bugs in GUI applications," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 107–116.

[61] A. M. Memon and M. B. Cohen, "Automated testing of gui applications: models, tools, and controlling flakiness," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1479–1480.

[62] D. Amalfitano, A. Fasolino, and P. Tramontana, "Experimenting a reverse engineering technique for modelling the behaviour of rich internet applications," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, 2009, pp. 571–574.

[63] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A GUI Crawling-Based Technique for Android Mobile Application Testing," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 252–261.

[64] A. C. R. Paiva, J. a. C. P. Faria, N. Tillmann, and R. A. M. Vidal, "A model-to-implementation mapping tool for automated model-based GUI testing," in *Proceedings of the 7th international conference on Formal Methods and Software Engineering*, ser. ICFEM'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 450–464.

[65] F. Zaraket, W. Masri, M. Adam, D. Hammoud, R. Hamzeh, R. Farhat, E. Khamissi, and J. Noujaim, "GUICOP: Specification-Based GUI Testing," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, 2012, pp. 747–751.

[66] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: high coverage, no false alarms," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 67–77.

[67] S. Staiger, "Reverse Engineering of Graphical User Interfaces Using Static Analyses," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, oct. 2007, pp. 189 –198.

[68] S. Ganov, C. Killmar, S. Khurshid, and D. E. Perry, "Event listener analysis and symbolic execution for testing GUI applications," in *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ser. ICFEM '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 69–87.

[69] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11.

[70] P. Mehlitz, O. Tkachuk, and M. Ujma, "JPF-AWT: Model checking GUI applications," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, 2011, pp. 584–587.

[71] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," in *Conference on Human factors in computing systems*, 2010, pp. 1535–1544.

[72] M. Sama, S. Elbaum, F. Raimondi, D. S. Rosenblum, and Z. Wang, "Context-Aware Adaptive Applications: Fault Patterns and Their Automated Identification," *IEEE Trans. Softw. Eng.*, vol. 36, no. 5, pp. 644–661, Sep. 2010.

[73] A. Swearngin, M. B. Cohen, B. E. John, and R. K. E. Bellamy, "Human performance regression testing," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 152–161.

[74] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 561–570.

[75] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 312–321.

[76] O. Sánchez Ramón, J. Sánchez Cuadrado, and J. García Molina, "Model-driven reverse engineering of legacy graphical user interfaces," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 147–150.

[77] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical User Interface (GUI) Testing: Systematic Mapping and Repository," *Information and Software Technology*, 2013.